Software Assurance Technology Center

SATC-TR-95-1001

SOFTWARE QUALITY METRICS for Object Oriented System Environments

JUNE 1995

National Aeronautics and Space Administration
Goddard Space Flight Center, Greenbelt Maryland 20771

TABLE OF CONTENTS

I. Introduction

II. Metrics Overview

III. Object-Oriented Overview

IV. Metrics for Object-Oriented Systems
- A. Metric Evaluation Criteria
- B. Traditional Metrics for Object-Oriented Systems
  - 1. Methods
    - Metric 1: Cyclomatic Complexity
    - Metric 2: Lines of Code
- C. Object-Oriented Specific Metrics
  - 1. Classes
    - a. Methods
      - Metric 3: Weighted Methods per Class (WMC)
    - b. Messages
      - Metric 4: Response for a Class (RFC)
    - c. Cohesion
      - Metric 5: Lack of Cohesion of Methods (LCOM)
    - d. Coupling
      - Metric 6: Coupling Between Object Classes (CBO)
  - 2. Inheritance
    - Metric 7: Depth of Inheritance Tree (DIT)
    - Metric 8: Number of Children (NOC)
- D. Example
- E. Summary
  - 1. Metric Summary
  - 2. COTS for Recommended Object-Oriented Metrics

V. Conclusions

Appendix A: Comprehensive Listing of Object-Oriented Metrics

Appendix B: Object-Oriented Detailed Discussion
- 1. Object
  - a. States/Attributes
  - b. Operations
  - c. Object Example
- 2. Class
- 3. Inheritance
- 4. Messages
- 5. Cohesion
- 6. Polymorphism
- 7. Object-Oriented Languages
- 8. Terminology

Appendix C: Cots Packages

Appendix D: Object-Oriented References

TABLES

Table 1: SATC Metrics for Object-oriented Systems
Table 2: Key Object-Oriented Definitions
Table 3: Summary of Recommended Object-Oriented Metrics
Table 4: Object-Oriented Metrics Supported by COTS

FIGURES

Figure 1: Pictorial Description of Key Object-Oriented Terms
Figure 2: Geometric Classes with Attribute, Operations and Methods
Figure 3: Pseudocode for Perimeter of an Equilateral Triangle
Figure 4: Notation of an Object
Figure 5: Sample Object
Figure 6: Conceptual View
Figure 7: Class with Objects
Figure 8: Inheritance Network
Figure 9: Message Passing

Software Quality Metrics for Object-Oriented System Environments

I. INTRODUCTION

Object-oriented analysis and design are popular concepts in today's software development environment. They are often heralded as the silver bullet for solving software problems,; while in reality there is no silver bullet, object-oriented has proved its value for systems that must be maintained and modified. Object-oriented software development requires a different approach from more traditional functional decomposition and data flow development methods. While the functional and data analysis approaches commence by considering the systems behavior and/or data separately; object-oriented analysis approaches the problem by looking or system entities that combine them. Object-oriented analysis and design focuses on objects as the primary agents involved in a computation; each class of data and related operations are collected into a single system entity.

The concepts of software metrics are well established, and many metrics relating to product quality have been developed and used. The SATC applies a model for evaluating software quality that has four goals:
(1) Stability of Requirements and Design, (2) Product Quality, (3) Testing Effectively, and
(4) Implementation Effectively. With object-oriented analysis and design methodologies gaining popularity, it is time to start investigating object-oriented metrics with respect to these goals. We are interested in the answer to the following questions:

- What concepts and structures in object-oriented affect the quality of the software?
- Can traditional metrics measure the critical object-oriented structures?
- If so, are the threshold values for the metrics the same for object-oriented designs as for functional/data designs?
- Which of the many new metrics found in literature are useful to measure the critical concepts in object-oriented?

This report summarizes results of the SATC's research on metrics for object-oriented systems. We start with a brief discussion of the metrics recommended by the SATC for object-oriented systems. These metrics include modifications of "traditional" metrics as well as "new" metrics for specific object-oriented structures. Since the object-oriented metrics require a cursory understanding of the object-oriented concepts, Section III presents a pictorial representation of the basic object-oriented structures and defines the key terms. A more extensive explanation of the object-oriented structures is in Appendix B and is referenced by Section III. In Section IV, we discuss the proposed object-oriented metrics with respect to the SATC Software Quality Model, specifically, their relationship to the attributes of quality (Goal 2: Product Quality -Structure/Architecture, Reuse, Maintainability). In the summary, we will address the availability of COTS packages to facilitate the collection of these metrics. Details on the COTS packages are given in Appendix C.

II. OVERVIEW - OBJECT-ORIENTED METRICS

In this report, the SATC documents its research into the current status of object-oriented metrics. The research was done by surveying the literature on object-oriented metrics then applying the SATC experience in traditional software metrics to select the object-oriented metrics that support the goal of measuring design and code quality. In addition, we required that a metric be feasible and have a clear relationship to the object-oriented structures being measured. At this time, many proposed object-oriented metrics lack a theoretical basis and have not yet been validated. Some of these metrics are too labor intensive to collect, or are too dependent on the implementation technology. The object-oriented metrics proposed by the SATC can be related to desirable software qualities.

The SATC's approach to identifying a set of object-oriented metrics was to focus on the primary, critical constructs of object-oriented design and to select metrics that apply to those areas. The suggested metrics are supported by most literature and some object-oriented tools. The metrics evaluate the object-oriented concepts: methods; classes; coupling; and inheritance. The metrics focus on internal object structure that reflects the complexity of each individual entity and on external complexity that measures the interactions among entities. The metrics measure computational complexity encompassing the efficiency of an algorithm and the use of machine resources, as well as psychological complexity factors that affect the ability of a programmer to create, modify, and comprehend software and the end user to effectively use the software.

We support the use of three traditional metrics and present six new metrics specifically for object-oriented systems. The SATC has found that there is considerable disagreement in the field about software quality metrics for object-oriented systems. Some researchers and practitioners contend traditional metrics are inappropriate for object-oriented systems. There are valid reasons for applying the traditional metrics however, if it can be done. The traditional metrics have been widely used, they are well understood by researchers and practitioners, and their relationships to software quality attributes have been validated. Table 1 presents an overview of the metrics proposed by the SATC for object-oriented systems. The SATC supports the continued use of traditional metrics, but within the structures and confines of object-oriented systems. The first three metrics in Table 1 are examples of how traditional metrics can be applied to the object-oriented structure of methods instead of functions or procedures. The next six metrics are specifically for object-oriented systems and the object-oriented construct applicable is indicated.

SOURCE	**METRIC**		**OBJECT-ORIENTED CONSTRUCT**
Traditional	CC	Cyclomatic complexity	Method
Traditional	SIZE	Lines of Code	Method
Traditional	COM	Comment percentage	Method
NEW Object-Oriented	WMC	Weighted methods per class	Class/Method
NEW Object-Oriented	RFC	Response for a class	Class/Message
NEW Object-Oriented	LCOM	Lack of cohesion of methods	Class/Cohesion
NEW Object-Oriented	CBO	Coupling between objects	Coupling
NEW Object-Oriented	DIT	Depth of inheritance tree	Inheritance
NEW Object-Oriented	NOC	Number of children	Inheritance

Table 1: SATC Metrics for Object-Oriented Systems

III. OVERVIEW - OBJECT-ORIENTED STRUCTURES

A brief description of the structure is given in this section using the pictorial description in Figure 1 and the definitions in Table 2. Appendix B contains a more comprehensive discussion of object-oriented concepts with additional diagrams of the structures. References to the detailed discussion in Appendix B is indicated by [ ] .

The new object-oriented development methods have their own terminology to reflect the new structural concepts. Referencing Figure 1, an object-oriented system starts by defining a *class* (Class A)[APP B.2] that contains related or similar *attributes* [APP B.1.1] and *operations* (some operations are *methods*) [APP B.1.2]. The classes are used as the basis for *objects* (Object A1) forming *hierarchical trees*. An object *inherits* all of the attributes and operations from its parent class, in addition to having its own attributes and operations.[APP B.3]. An object can also become a class for other objects (Object A1 -> Class B), forming another branch in the hierarchical tree structure. When an object is applied and contains data or information, it is an *instantiation* of the object. Objects interact or communicate by passing *messages* [APP B.4]. When a message is passed between two objects, the object are *coupled*. These specific terms are defined in Table 1; a more comprehensive listing is contained in Appendix B.

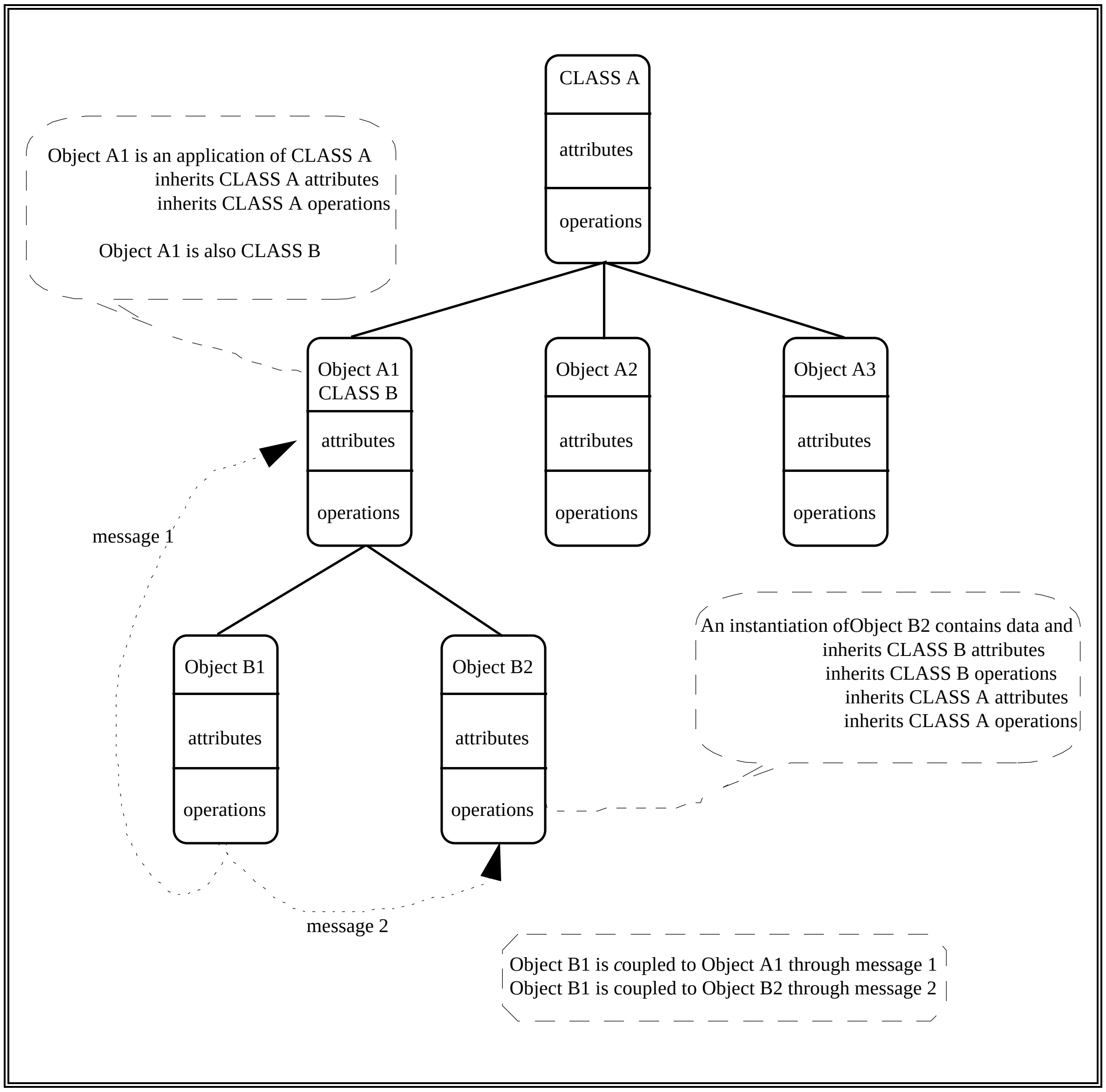
CLASS A
attributes
operations
Object A1 is an application of CLASS A
inherits CLASS A attributes
inherits CLASS A operations
Object A1 is also CLASS B
Object A1 CLASS B
attributes
operations
Object A2
attributes
operations
Object A3
attributes
operations
message 1
Object B1
attributes
operations
Object B2
attributes
operations
An instantiation ofObject B2 contains data and
inherits CLASS B attributes
inherits CLASS B operations
inherits CLASS A attributes
inherits CLASS A operations
message 2
Object B1 is coupled to Object A1 through message 1
Object B1 is coupled to Object B2 through message 2

Figure 1: Pictorial Description of Object-Oriented Terms

Class	A set of objects that share a common structure and common behavior manifested by a set of methods; the set serves as a template from which objects can be instantiated (created).[APP B.2]
Cohesion	The degree to which the methods within a class are related to one another. [APP B.5]

Coupling	Object X is coupled to object Y if and only if X sends a message to Y. [APP B.4]
Inheritance	A relationship among classes, wherein one class shares the structure or methods defined in one (for single inheritance) or more (for multiple inheritance) other classes. [APP B.3]
Instantiation	The process of creating an instance of the object and binding or adding the specific data. [APP B.3]
Message	A request that an object makes of another object to perform an operation. [APP B.4]
Method	An operation upon on object, defined as part of the declaration of a class. Methods are operations, but not all operations are actual methods declared for a specific class. [APP B.1.2]
Object	An instantiation of some class which is able to save a state (information) and which offers a number of operations to examine or affect this state. [APP B.1]

Table 2: Key Object-Oriented Terms for Metrics

IV. METRICS FOR OBJECT-ORIENTED SYSTEMS

A. Metric Evaluation Criteria

While metrics for the traditional functional decomposition and data analysis design approach deal with the design structure and/or data structure independently, object-oriented metrics must be able to focus on the combination of function and data as an integrated object. The evaluation of the utility of a metric as a quantitative measure of software quality must relate to the SATC Software Quality Model, although we feel strongly we have selected metrics that are useful in a wide range of models. The second goal of the SATC model is the measurement of Product (Code) Quality. The metrics selected are applicable to the attributes that support this goal. The object-oriented metric criteria, therefore, are the evaluation of the following areas:

- Efficiency of the implementation of the design **-** Were the constructs efficiently designed?
- Complexity **-** Could the constructs be used more effectively to decrease the architectural complexity?
- Understandability/Usability **-** Does the design increase the psychological complexity?
- Reusability/Application specific **-** Is the design application specific?
- Testability/Maintenance **-** Does the structure enhance testing?

Whether a metrics is “traditional” or “new”, it must be effective in measuring on of these areas. As each metric is presented, we will briefly discuss its applicability to these areas.

B. Traditional Metrics

B.1: Methods
In an object-oriented system, traditional metrics are generally applied to the methods that comprise the operations of a class. A method is a component of an object that operates on data in response to messages and is defined as part of the declaration of a class [APP B.1.2]. Methods reflect how a problem is broken up and the capabilities other classes expect of a given class. Two traditional metrics are discussed here: cyclomatic complexity and line counts (size).

- METRIC 1: Cyclomatic Complexity (CC)

The cyclomatic complexity (McCabe) is used to evaluate the application of an algorithm. A method with a low cyclomatic complexity may imply that decisions are deferred through message passing, not that the methods is not complex. The cyclomatic complexity cannot be used to measure the complexity of a class because of inheritance, but the cyclomatic complexity of individual methods can be combined with other measures to evaluate the complexity of the class. Although this metric is specifically applicable to the evaluation of complexity, it also is related to all of the other attributes.

- METRIC 2: Line Count ==> Size/Documentation

  Various line counts are also applied to methods. These include counting all physical lines of code, the number of statements and the number comment lines. Thresholds for evaluating the meaning of size measures may have to vary greatly depending on the coding language. However, since size limitations are based on ease of understanding by the developers and maintainers, routines of large size will always pose a higher risk in attributes such as Understandability, Reusability, and Maintainability. This metric can be used to evaluate all the attributes, but most often is a measure of Understandability, Reusability, and Maintainability.

- METRIC 3: Comment Percentage

  The Line Count metric can be expanded to include a count of the number of comments, both on-line and stand-alone. The comment percentage is calculated by the total number of comments divided by the total lines of code less the number of blank lines. Since comments assist developers and maintainers, this metric is used to evaluate the attributes of Understandability, Reusability, and Maintainability.

C. Object-Oriented Specific Metrics

As indicated in Appendix A, many different metrics have been proposed for object-oriented systems. The object-oriented metrics that were chosen measure principle structures that, if they are improperly designed, negatively affect the design and code quality attributes.

The selected object-oriented metrics are primarily applied to the concepts of classes, coupling, and inheritance. Preceding each metric, a brief description of the object-oriented structure is given with references to Appendix B. For some of the object-oriented metrics discussed here, multiple definitions are given. As with traditional metrics, researchers and practitioners have not reached a common definition or counting methodology. In some cases, the counting method for a metric is determined by the software analysis package being used to collect the metrics.

C.1 Class

A class is a template from which objects can be created. This set of objects share a common structure and a common behavior manifested by the set of methods [APP B.2, Figure 5]. Three class metrics described here measure the complexity of a class using the class’s methods, messages and cohesion.

C.1.1 Method

A method is an operation upon an object [APP B.1.2].

- METRIC 4: Weighted Methods per Class (WMC)

  The WMC is a count of the methods implemented within a class or the sum of the complexities of the methods (method complexity is measured by cyclomatic complexity). The second measurement is difficult to implement since not all methods are assessable within the class hierarchy due to inheritance. The number of methods and the complexity of the methods involved is a predictor of how much time and effort is required to develop and maintain the class. The larger the number of methods in a class, the greater the potential impact on children since children will inherit all the methods defined in a class. Classes with large numbers of methods are likely to be more application specific, limiting the possibility of reuse. This metric measures usability and reusability.

C.1.2 Message

A message is a request that an object makes of another object to perform an operation. The operation executed as a result of receiving a message is called a method. The next metric looks at methods and messages within a class [APP B.4].

- METRIC 5: Response for a Class (RFC)

  The RFC is the cardinality of the set of all methods that can be invoked in response to a message to an object of the class or by some method in the class. This includes all methods accessible within the class hierarchy. This metric looks at the combination of the complexity of a class through the number of methods and the amount of communication with other classes. The larger the number of methods that can be invoked from a class through messages, the greater the complexity of the class. If a large number of methods can be invoked in response to a message, the testing and debugging of the class becomes complicated since it requires a greater level of understanding on the part of the tester. A worst case value for possible responses will assist in the appropriate allocation of testing time. This metric evaluations system design as well as the usability and the testability.

C.1.3 Cohesion

Cohesion is the degree to which methods within a class are related to one another and work together to provide well-bounded behavior [APP B.5]. Effective object-oriented designs maximize cohesion since it promotes encapsulation. The third class metric investigates cohesion.

- METRIC 6: Lack of Cohesion of Methods (LCOM)

  LCOM measures the degree of similarity of methods by instance variable or attributes. Any measure of separateness of methods helps identify flaws in the design of classes. There are at least two different ways of measuring cohesion:
  1. Calculate for each data field in a class what percentage of the methods use that data field. Average the percentages then subtract from 100%. Lower percentages mean greater cohesion of data and methods in the class.
  2. Methods are more similar if they operate on the same attributes. Count the number of disjoint sets produced from the intersection of the sets of attributes used by the methods.

  High cohesion indicates good class subdivision. Lack of cohesion or low cohesion increases complexity, thereby increasing the likelihood of errors during the development process. Classes with low cohesion could probably be subdivided into two or more subclasses with increased cohesion. This metric evaluates the design implementation s well as reusability.

C.1.4 Coupling

Coupling is a measure of the strength of association established by a connection from one entity to another. Classes (objects) are coupled three ways:

1. When a message is passed between objects, the objects are said to be coupled [B4, Figure 7].

2. Classes are coupled when methods declared in one class use methods or attributes of the other classes.
3. Inheritance introduces significant tight coupling between superclasses and their subclasses [APP B.3]. (Since good object-oriented design requires a balance between coupling and inheritance, coupling measures focus on non-inheritance coupling.)

The next object-oriented metric measures coupling strength.

- METRIC 7: Coupling Between Object Classes (CBO)

  CBO is a count of the number of other classes to which a class is coupled. It is measured by counting the number of distinct non-inheritance related class hierarchies on which a class depends. Excessive coupling is detrimental to modular design and prevents reuse. The more independent a class is, the easier it is reuse in another application. The larger the number of couples, the higher the sensitivity to changes in other parts of the design and therefore maintenance is more difficult. Strong coupling complicates a system since a module is harder to understand, change or correct by itself if it is interrelated with other modules. Complexity can be reduced by designing systems with the weakest possible coupling between modules. This improves modularity and promotes encapsulation. CBO evaluates design implementation and reusability.

C.2 Inheritance
Another design abstraction in object-oriented systems is the use of inheritance. Inheritance is a type of relationship among classes that enables programmers to reuse previously defined objects including variables and operators [APP B.3, Figure 6]. Inheritance decreases complexity by reducing the number of operations and operators, but this abstraction of objects can make maintenance and design difficult. The two metrics used to measure the amount of inheritance are the depth and breadth of the inheritance hierarchy.

- METRIC 8: Depth of Inheritance Tree (DIT)

  The depth of a class within the inheritance hierarchy is the maximum length from the class node to the root of the tree and is measured by the number of ancestor classes. The deeper a class is within the hierarchy, the greater the number methods it is likely to inherit making it more complex to predict its behavior. Deeper trees constitute greater design complexity, since more methods and classes are involved, but the greater the potential for reuse of inherited methods. A support metric for DIT is the number of methods inherited (NMI). This metric primarily evaluates reuse but also relates to understandability and testability.

- METRIC 9: Number of Children (NOC)

  The number of children is the number of immediate subclasses subordinate to a class in the hierarchy. It is an indicator of the potential influence a class can have on the design and on the system. The greater the number of children, the greater the likelihood of improper abstraction of the parent and may be a case of misuse of subclassing. But the greater the number of children, the greater the reuse since inheritance is a form of reuse. If a class has a large number of children, it may require more testing of the methods of that class, thus increase the testing time. NOC, therefore, primarily evaluates testability and design.

D. Object-Oriented Metrics Example

Object-Oriented design requires a different way of thinking. In order to demonstrate the concepts of object-oriented and how the suggested metrics could be applied, the example in Figure 2 was developed. Convex Set, which is the set of all geometric figures, is a superclass. Convex Set has an operation to draw the specified figures and uses drawing methods found in another super set not shown here. From this, we can develop two subclasses, polygon and ellipse; these become classes since they have children objects that are subclasses. Polygon contains a method to calculate the perimeter. This equation is accessible to all child objects. Following the hierarchy to equilateral, through inheritance we know it is a closed figure (convex set) of straight lines (polygon) with 3 sides and sum of the interior angles is 180 (triangle). If we create an equilateral triangle with the side length of 5, it is called an instantiation of the equilateral object.

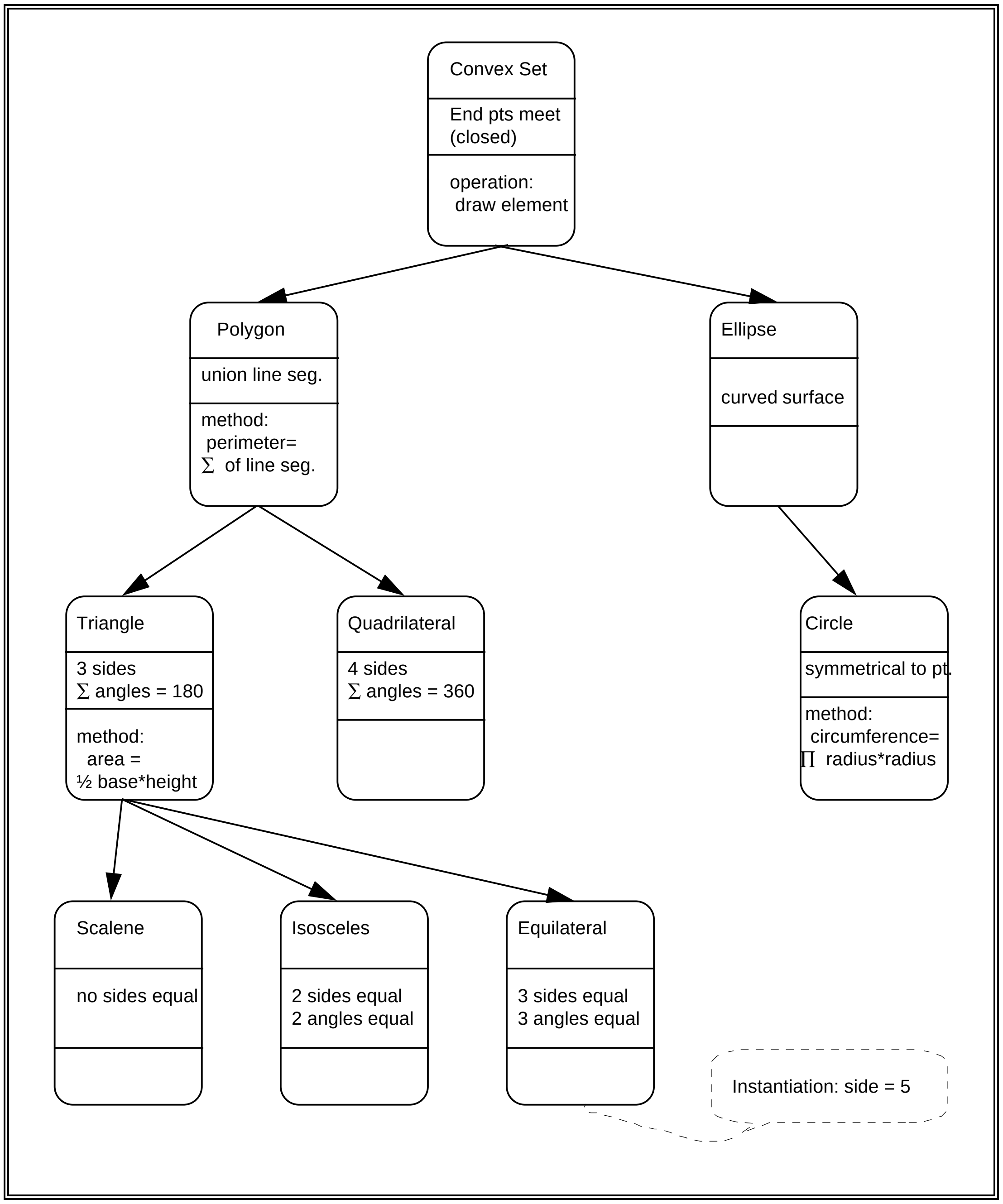

Figure 2: Geometric Classes with Attributes, Operations and Methods

The eight metrics discussed in Section IV.C can now be applied to this example. Since the object EQUILATERAL is a child of TRIANGLE/POLYGON, the formula for the perimeter is accessible. (A message is used to transfer the value to the equation so the perimeter can be calculated.) From TRIANGLE, the attribute of 3 sides is inherited. Since EQUILATERAL has the attribute that all sides are equal, the equation for the perimeter will be modified to 3 times the length of the side. Before applying the equation, an instantiation of the TRIANGLE object must be created. This is where the actual value(s) are assigned, and the length of the side, X, is stored. The pseudocode for the method to determine the perimeter of an equilateral triangle in Figure 3.

```
STEPS:

Create instantiation of equilateral triangle
Assign the length of the side ==> X
Send message to class POLYGON for perimeter equation

  If X > 0                      {* hence a valid measurement *}
     then
        perimeter = 3*X             {* inherited from polygon *}
     else send error message
```

Figure 3: Method for perimeter of equilateral triangle

- METRIC 1: Cyclomatic Complexity
  This is an evaluation of the algorithm using a count of the number of test paths. There is one vertical path that spits into a second possible path, giving the method a cyclomatic complexity of 2.

- METRIC 2: Count of Lines
  For the pseudocode, the easiest line count is by executable statements, which is 6, total lines of code is 8, with 1 blank and 2 comments.

- METRIC 3: Comment Percentage

Comment percentage = 2/(8-1) = 29%

- METRIC 4: Weighted Methods per Class (WMC)
  This metric counts the number of methods per class, so each class will have a different value for WMC. We can reference Figure 2 to evaluate the WMC for two classes below:
  Class: POLYGON = count of number of methods = 2
  Class: QUADRILATERAL = 1

  If text was given for each method, we could sum their cyclomatic complexity within the class, allowing us to weight the methods for a more realistic evaluation.

- METRIC 5: Response for a Class (RFC)
  For RFC we need the to sum the number of methods that can be invoked in response to a message, including all methods accessible within the class hierarchy. Using the POLYGON class, for perimeter, 5 objects could invoke that method, and the method area (in TRIANGLE) could invoke 3 messages, so the RFC for POLYGON is 8.

  Class: POLYGON = number of messages for method (perimeter) = 5 +
  number of messages for method (area) = 3 ==> RFC = 8

- METRIC 6: Lack of Cohesion of Methods (LCOM)
  We can apply this metric to two different classes for examples of low and higher cohesion. Looking at the degree of similarity between POLYGON and ELLIPSE, we observe that the intersection of the sets of attributes of each is a disjoint set, there are no attributes in common. (POLYGON = {union of line segments};   ELLIPSE = {curved surface}) This implies low cohesion between these classes and the design might be better if these classes were not related through CONVEX SET. Looking at the classes TRIANGLE and QUADRILATERAL, we observe their attributes are the same, {sides, angles}, forming a common set and implying high cohesion and a good design.

- METRIC 7: Coupling between Object Classes (COB)
  COB is measured by counting the number of non-inherited related class hierarchies on which a class depends. For the class *Polygon*, all coupling is within the class (messages to inherited methods do not count), so the COB = 0. If we look at the class *Convex Set*, the operation Draw Element, this links to outside the class where the drawing software resides, implying high coupling and suggesting this operation and class should be separate. (Note - this separation was also suggested by Metric 5, LCOM.)

- METRIC 8: Depth of Inheritance Tree (DIT)
  The depth is measured by the maximum length from the class node to the root of the tree. For POLYGON, DIT = 2, it is 1 level below the CONVEX SET. For TRIANGLE, DIT = 1.

- METRIC 9: Number of Children (NOC)
  This metric is a count of the number of children of a class. For the class POLYGON, NOC = 2; for TRIANGLE, NOC = 3.

D. Summary

D.1 Metrics Summary

In addition to assessing the software attributes related to software quality as specified in Section IV.A, software metrics should meet certain theoretical criteria. These criteria are specified in terms of the object-oriented structures to which the metrics are to be applied.

- Noncoarseness - Not every class can have the same value for a metric, otherwise it has lost its value as a measurement.
- Nonuniqueness (notion of equivalence) - Two classes can have the same metric value (i.e., two classes are equally complex).
- Design details are important - Even though two class designs perform the same function, the details of the design matter in determining the metric for the class.
- Monotonicity - The metric for the combination of two classes can never be less than the metric for either of the component classes.
- Nonequivalence of interaction - The interaction between two classes can be different between two other classes resulting in different complexity values for the combination.
- Interaction increases complexity - When two classes are combined, the interaction between classes can increase the complexity metric value.

Although the proposed metrics meet these criteria, we will not demonstrate that in this report.

Table 2 summarizes the eight metrics recommended for object-oriented systems. They cover the key concepts for object-oriented designs: methods, classes (cohesion), coupling, and inheritance.

Metrics for object-oriented systems is a relatively new field of study. Although some numeric thresholds are suggested by COTS developers, there is little (if any) application data to justify them. Table 2 instead gives a general interpretation for the metrics (e.g. larger numbers denote application specificity).

METRIC		OBJECT-ORIENTED FEATURE	CONCEPT	MEASUREMENT METHOD	INTERPRETATION
CC	Cyclomatic complexity	Method	Complexity	# algorithmic test paths	Low => decisions deferred through message passing Low not necessarily less complex
SIZE	Lines of Code	Method	Complexity	# physical lines, statements, and/or comments	Should be small
COM	Comment Percentage	Method	Usability Reusability	# comments divided by the total line count less # blank lines	~ 20 - 30 %
WMC	Weighted methods per class	Class/Method	Complexity Usability Reusability	1) # methods implemented within a class 2) Sum of complexity of methods	Larger => greater potential impact on children through inheritance; application specific
RFC	Response for a class	Class/Method	Design Usability Testability	# methods invoked in response to a message	Larger => greater complexity and decreased understandability; testing and debugging more complicated
LCOM	Lack of cohesion of methods	Class/Cohesion	Design Reusability	Similarity of methods within a class by attributes	High => good class subdivision Low => increased complexity - subdivide
CBO	Coupling between objects	Coupling	Design Reusability	# distinct non-inherited related classes inherited	High => poor design; difficult to understand; decreased reuse; increase maintenance
DIT	Depth of inheritance tree	Inheritance	Reusability Understandability testability	Maximum length from class node to root	Higher => more complex; more reuse
NOC	Number of children	Inheritance	Design	# immediate subclass	Higher => more reuse; poor design increasing testing

Table 2: Object-oriented Metrics Summary

D.2 COTS Support

As noted earlier, some of the object-oriented metrics, although producing valuable insights to the software, may be difficult to collect electronically. The SATC surveyed approximately 12 object-oriented software analysis COTS to determine which of the suggested object-oriented metrics were supported. Some packages advertise object-oriented metrics, but only traditional metrics were applied with no alterations. Although most of the actual object-oriented packages offered a variety of metrics, only the eight metrics proposed in this report are noted. Some of the packages used different names for the metrics, but the description matched. As denoted in the chart, some packages used diagrams to indicate the metrics, other metrics were not specified but could be derived. The complete list of packages surveyed is in Appendix D.

TOOL	CC	SIZE	COM	WMC	RFC	LCOM	CBO	DIT	NOC
ADAMET	N	N	C				?	C	C
Flexsys	N	N	C	N	N	N	?	N	N
McCabe	N	N	N	N	N	N	N	N	N
ParaSET				D	D	D		D	D
PR:QA	?			N	N	N		N	N
UX Metrics	N	N	N	C	C	C	?	N	N

N = Number provided
C = Can be calculated
D = Diagram only

Table 4: COTS Support Object-Oriented Metrics

V. CONCLUSIONS

The SATC has proposed nine metrics for object-oriented systems. They cover the key concepts for object-oriented designs: methods, classes (cohesion), coupling, and inheritance. The metrics are to be applied to the evaluation of the following software qualities:

- Efficiency of the implementation of the design **-** Were the constructs efficiently designed?
- Complexity **-** Could the constructs be used more effectively to decrease the architectural complexity?
- Understandability/Usability **-** Does the design increase the psychological complexity?
- Reusability/Application specific **-** Is the design application specific?
- Testability/Maintenance **-** Does the structure enhance testing?

Future work will be to define criteria for the metrics. That is, acceptable ranges for each metric will have to developed, based on the effect of the metric on desirable software qualities. The SATC will first develop a database of the metrics from actual code, to understand a “normal” range, and then evaluate the impact on system reliability and maintainability of deviations from that range. (**NOTE**: the above may be overly simplistic.)

APPENDIX A: COMPREHENSIVE LISTING OF OBJECT-ORIENTED METRICS

	METRIC	**STRUCTURE**
ACM	attribute complexity metric	Class
CBO	coupling between object classes	Coupling
CC	McCabe's cyclomatic complexity	Method
CC	class complexity	Coupling
CC2	progeny count	Class
CC3	parent count	Class
CCM	class coupling metric	Coupling
CCO	class cohesion	Class
CCP	class coupling	Coupling
CM	cohesion metric	Class
DAC	data abstraction coupling	Class
DIT	depth of inheritance tree	Inheritance
FAN	fan-in	Class
FFU	friend function	Class
FOC	function- oriented code	Class
GUS	global usage	Class
HNL	hierarchy nesting level	Inheritance
IVU	instance variable usage	Class
LCOM	lack of cohesion of methods	Class
LOC	lines of code	Method
MCX	method complexity	Method
MPC	message passing coupling	Coupling
MUI	multiple inheritance	Inheritance
NCM	number of class methods	Class
NCV	number of class variables	Class
NIM	number of instance methods	Class
NIV	number of instance variables	Class
NMA	number of methods added	Inheritance
NMI	number of methods inherited	Inheritance
NMO	number of methods overridden	Inheritance
NOC	number of children	Inheritance
NOM	number of message sends	Method
NOM	number of local methods	Class
NOT	number of tramps	Coupling
OACM	operation argument complexity metric	Class
OCM	operation coupling metric	Coupling
OXM	operation complexity metric	Class
PIM	number of public instance methods	Class
PPM	parameters per methods	Class
RFC	raw function counts	Class
RFC	response for a class	Class
SIX	specialization index	Inheritance
SIZE1	language dependent delimiter	Method
SIZE2	number of attributes + number of local methods	Class
SSM	Halstead software science metrics	Method
VOD	violations of the Law of Demeter	Coupling
WAC	weighted attributes per class	Class
WMC	weighted methods per class	Class

APPENDIX B: OBJECT-ORIENTED ANALYSIS AND DESIGN

APP B.1 Objects

An object-oriented system is an organized collection of cooperative objects representing real world entities. Amongst the most prominent qualities of an object-oriented design system are:

- Understanding of the system is easier as the semantic between the system and reality is small.
- Modifications to the model tend to be local as they often result from an individual item, represented by a single object.
- Reuse is often enhanced through the independence of the items.

Object-oriented design interconnects data items (**objects**) and processing operations (**messages**) such that information and processes are modularized.

An **object** is an entity which has a **state/attributes** (whose representation is hidden) and a defined set of **operations** which operate on that state. Objects are initially identified by examining the problem statement or by grammatically parsing the system description and identifying each noun or noun clause as possible objects. If an object is required to implement a solution, then it is part of the solution space; otherwise, if an object is necessary only to describe a solution, it is part of the problem space.
The notation used to represent an object is shown in Figure 4.

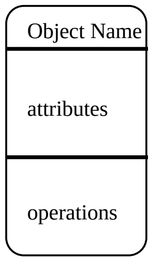

Figure 4 - Notation for an Object

APP B.1.1 State/Attributes

The **state** is represented as a set of object **attributes**. Attributes describe and define an object by clarifying what is meant by the object in the context of the problem space. To develop a meaningful set of attributes for an object, the analyst can refer to the system description and select those things that reasonably “belong” to the object. The following question should be answered for each object: What data items fully define this object in the context of the problem at hand?

APP B.1.2 Operations

The **operations** contain control and procedural constructs that may be invoked by a message - a request to the object to perform one of its operations. An operation changes an object in some way; specifically, it changes one or more attribute values that are contained within the object. Operations can generally be divided into three broad categories: 1) operations that manipulate data in some way; 2) operations that perform a calculation; 3) operations that monitor an object for the occurrence of a controlling event. Operations are identified by examining all verbs stated in the processing narrative within the system description.

APP B.1.3 Object Example

Figure 5 shows an example of an object. The name of the Object is *Chair*, it has the attributes of *cost*, *dimensions, weight, location* and *color*, and the operations are *buy*, *sell*, *weigh*, and *move*.

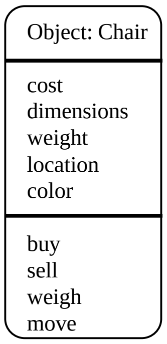

Figure 5 - Sample Object

The object's behavior and information are stored within the object (**encapsulated**) and can only be manipulated when the object is ordered to perform operations. This **encapsulation** supports **information hiding**. The attributes that are visible to the outside (e.g., cost) are the **specification part**, but each object also has a **private** or **body part** that is hidden from the outside. At the design stage, the implementation details of the private part are not yet specified, these usually are added at runtime (e.g. chair cost = $45.32). The designer's conception of the object *chair* may be as shown in Figure 6; in actuality, the private part is never shown.

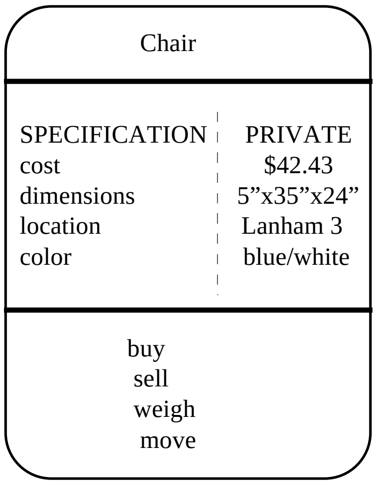

Figure 6: Conceptual View of an Object's Attributes

APP B.2 Class

Chair is a member (the term **instance** is commonly used) of a much larger **class** of objects called *Furniture*. A set of generic attributes can be associated with every object in the class of *furniture*. Because ***chair*** is a member of the class of *furniture*, the *chair* **inherits** all attributes defined for the class. Once the class has been defined, the attributes can be reused when new instances of the class are created, such as for the object *table*. These concepts are demonstrated in Figure 7. In the figure, the inherited attributes for *chair* and *table* are re-written in each object, normally they are assumed and not listed.

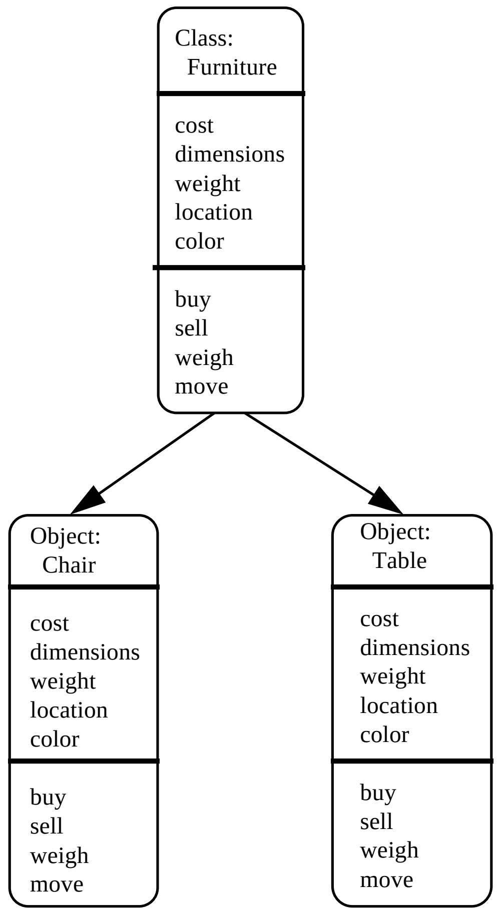

Figure 7: Class Furniture with Objects and Inherited Attributes and Operations

APP B.3 Inheritance

Object **instatiation** is when an object inherits the attributes and operations of their class. Object classes can also be objects (called **super-classes**) and **inheritance networks** can be established. An **inheritance hierarchy or tree** is created when a class can inherit attributes from a single **super class** as demonstrated in Figure 8. Note inherited attributes are not shown. *Project manager* has all attributes of *Manager* and *Employee* (super-class) but they are not repeated and the state representation is hidden (i.e. specifically information such as *address*).

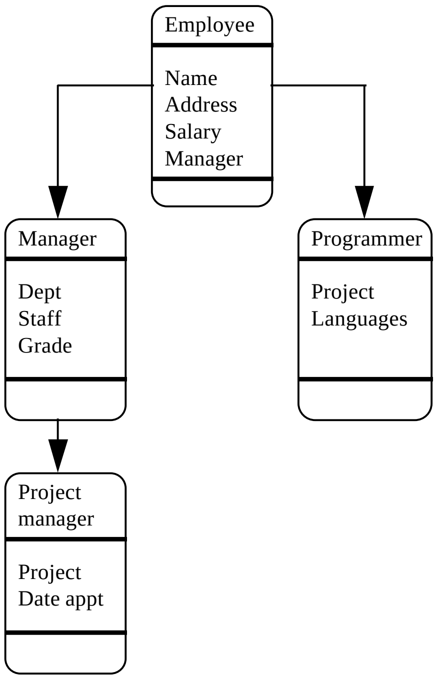

Figure 8: Inheritance Network

Inheritance is a form of reuse and can decrease the complexity by reducing the number of operations and operands. It is also an abstraction technique and provides classification information about system entities. Inheritance is invaluable in prototyping as it permits code reuse and allows rapid changes to be made to an object without side effects which corrupt other parts of the system. However, constructing a consistent inheritance hierarchy without duplicating attributes, and attributes and operations at the right level of abstraction, is difficult. Although inheritance can be a useful abstraction technique to show relationships between the design types, but it is not essential in object-oriented design. Some designers prefer to apply inheritance during implementation instead of design to reduce misunderstanding when reading the design; without inheritance, all operations and attributes of an object class must be specifically stated.

Encapsulation and inheritance are both essential within object-oriented programming; they are somewhat incompatible with each other however. Encapsulation means that the one object using a class should not see its internal representation. But if we regard a descendant of a class as a user of the class, then the user has complete access to the internal parts of the class (through inheritance). This contradiction is due to the fact that we have three types of user: those who use the class via its interface, those who use the class through inheritance, and those who actually implement the class. In C++, therefore, three possibilities for encapsulation of operations and data structures are used: “public” means that all three user types can access the class operations, “protected” means that only the class itself or descendants of the class can access the parts of the class, “private” means that only the class itself can access the parts. Of course, there three types are combined when a class is developed.

APP B.4 Messages

Objects communicate by passing messages to each other and these messages initiate object operations. A message usually is implemented as a procedure or function call. It has two parts: 1) the name of the service requested by the call object; 2) copies of the information (usually in the form of variables) from the calling object needed to execute the service and the holder of the results. Communication by exchanging messages rather than shared variables eliminates shared data areas. This reduces overall system **coupling** as there is no possibility of unexpected modifications to shared information. The receiver (object) responds to the message by first choosing the operation that implements the message name, executing this operation, then returning control to the caller.

Using Figure 9, four objects *A*, *B*, *C*, and *D*, communicate with one another by passing messages. For example, if object *B* required processing associated with operation *op10* of object *D*, it would send *D* a message that would take the form:

message: (destination, operation, arguments)

where "destination" defines the object (in this case, object *D*) to receive the message, "operation" refers to the operation that is to receive the message (*op10*), and the "arguments" provides information that is required for the operation to be successful. This message **couples** object *B* and Object *D*. As part of the execution of *op10*, object *D* may send a message to object *C* of the form:

message: (C, op8, <data>)

C finds *op8*, performs it, and then returns control to *D*. Operation *op10* completes and control is returned to *B*.

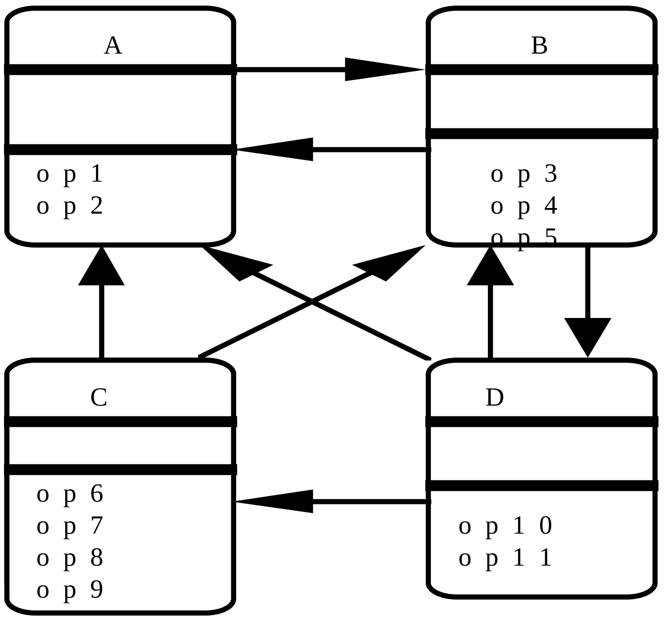

Figure 9: Message Passing

APP B.5 Cohesion

Cohesion refers to the internal consistency within the parts of the design. Cohesion is centered on data that is encapsulated within an object and on how methods interact with data to provide well-bounded behavior. Degree of similarity of methods is a major aspect of object class cohesiveness. The objective is to achieve maximum cohesion. Programs that are adaptable and reusable are low in coupling and high in cohesion.

APP B.6 Polymorphism

Polymorphism is a very important concept that allows flexible systems to be built. This concept allows developers to specify what shall occur and not how it shall occur. Polymorphism means having the ability to take several forms. For object-oriented systems, polymorphism allows the implementation of a given operation to be dependent on the object that contains the operation; an operation can be implemented in different ways in different classes. The sender instance of the message does not need to know the receiving instance class. The sender provides only a request for a specific operation, while the receiver knows how to perform the operation. The message can be interpreted different ways, dependent on the receivers class. It is, therefore, the instance which receives the message that determines its interpretation, and not the transmitting instance. Polymorphism assist programmers to reduce complexity because:

- Programmers do not have to comprehend or even be aware of existing operations to add a new operation.
- Programmers do not have to consider methods that are part of the object when naming the operation
- Programmers can preserve the semantics of the operation within the object and provide common interfaces to types of objects that are similar.

APP B.7 Object-Oriented Languages

Object-oriented design is not the same as object oriented programming. Object-oriented languages specifically support the features of object-oriented design, but the design can be implemented in any language. C++ and Smalltalk are object-oriented languages, specifically supporting all constructs. Ada is not an object-oriented language, it does not support polymorphism and inheritance but can be used for object-oriented designs.

APP B.8 Terminology

Aggregate object (aggregation) - An object composed of two or more other objects. An object that is *part of* two or more other objects.

Attribute - A variable or parameter that is encapsulated into an object.

Class - A set of objects that share a common structure and common behavior manifested by a set of methods; the set serves as a template from which objects can be created.

Class Structure - A graph whose vertices rep[resent classes and whose arcs represent relationships among these classes.

Cohesion - The degree to which the methods within a class are related to one another.

Collaborating classes - If a class sends a message to another class, the classes are said to be collaborating.

Coupling - Object X is coupled to object Y if and only if X sends a message to Y.

Encapsulation - The process of bundling together the elements of an abstraction that constitute its structure and behavior.

Information hiding - The process of hiding the structure of an object and the implementation details of its methods. An object has a public interface and a private representation; these two elements are kept distinct.

Inheritance - A relationship among classes, wherein one class shares the structure or methods defined in on (for single inheritance) or more (for multiple inheritance) other classes.

Instance - An object with specific structure, specific methods, and an identity.

Instantiation - The process of filling in the template of a generic class to produce a class from which one can create instances.

Message - A request that an object makes of another object to perform an operation.

Method - An operation upon on object, defined as part of the declaration of a class. Methods are operation, but not all operations are actual methods declared for a specific class.

Object - An instantiation of some class which is able to save a state (information) and which offers a number of operations to examine or that affect this state.

Polymorphism - The ability of an object to interpret a message differently at execution depending upon the superclass of the calling object.

Superclass - The class from which another class inherits its attributes and methods.

APPENDIX C: METRIC PACKAGES

AdaMET, Dynamics Research Corporation, Andover, MA.	
Checkpoint, Software Productivity Research, Burlington, MA.	(No Object-oriented metrics)
DecisionVision, Software Business Management, Westford, MA.	(No Object-oriented metrics)
Logicore Software Development Environment, Logicon, Arlington, VA.	(No Object-oriented metrics)
MetKit, Bramer Ltd., Fleet, Hants, UK.	(No Object-oriented metrics)
McCabe Object-Oriented Tool, McCabe & Associates, Columbia, MD.	
ParaSET, Software Emancipation Technology, Inc., Lexington, MA.	
PR:QA, ASTA Incorporated, Nashua, NH.	
Rational Environment, Rational, Bethesda, MD.	(No Object-oriented metrics)
Spiders-3, Statistica, Inc., Rockville, MD.	(No Object-oriented metrics)
UX-Metrics, Set Laboratories Inc., Mulino, OR.	

APPENDIX D: OBJECT-ORIENTED REFERENCES

Booch, Grady, *Object-oriented Analysis and Design with Applications*, The Benjamin/Cummings Publishing Company, Inc., 1994.

Chidamber, Shyam and Kemerer, Chris, "A Metrics Suite for Object-Oriented Design", *IEEE Transactions on Software Engineering*, June, 1994, pp. 476-492.

Hudli, r., Hoskins, C., Hudli, A., "Software Metrics for Object-oriented Designs", IEEE, 1994.

Jacobson, Ivar, *Object-oriented Software Engineering, A Use Case Driven Approach*, Addison-Wesley Publishing Company, 1993.

Lee, Y., Liang, B., Wang, F., "Some Complexity Metrics for Object-Oriented Programs Based on Information Flow", *Proceedings: CompEuro*, March, 1993, pp. 302-310.

Lorenz, Mark and Kidd, Jeff, *Object-Oriented Software Metrics*, Prentice Hall Publishing, 1994.

Pressman, Roger S., *Software Engineering, A Practitioner's Approach*, McGraw-Hill Publishing, 19xx.

Sommerville, Ian, *Software Engineering*, Addison-Wesley Publishing Company, 1992.

Sharble, Robert, and Cohen, Samuel, "The Object-Oriented Brewery: A Comparison of Two Object-Oriented Development Methods", *Software Engineering Notes*, Vol 18, No 2., April 1993, pp 60 -73.

Stinson, Michael, and Archer, Clark, "Object-Oriented Software Measures: The State of the Art", *Proceedings: Software Technology Conference*, April, 1995.

Tegarden, D., Sheetz, S., Monarchi, D., "Effectiveness of Traditional Software Metrics for Object-Oriented Systems", *Proceedings: 25th Hawaii International Conference on System Sciences*, January, 1992, pp. 359-368.

Williams, John D., "Metrics for Object-Oriented Projects", *Proceedings: ObjectExpoEuro Conference*, July, 1993, pp. 13-18.